On Longest Repeat Queries☆

Atalay Mert İleri[a], M. Oğuzhan Külekci[b], Bojian Xu[c,*]

[a] *Massachusetts Institute of Technology, USA*
[b] *Istanbul Medipol University, Turkey*
[c] *Eastern Washington University, USA*

Abstract

Repeat finding in strings has important applications in subfields such as computational biology. Surprisingly, all prior work on repeat finding did not consider the constraint on the locality of repeats. In this paper, we propose and study the problem of finding longest repetitive substrings covering particular string positions. We propose an $O(n)$ time and space algorithm for finding the longest repeat covering every position of a string of size n. Our work is optimal since the reading and the storage of an input string of size n takes $O(n)$ time and space. Because any substring of a repeat is also a repeat, our solution to longest repeat queries effectively provides a "stabbing" tool for practitioners for finding most of the repeats that cover particular string positions.

Keywords: information retrieval, string processing, repeats, regularities, repetitive structures

1. Introduction

Repetitive structures and regularities finding in genomes and proteins is important as these structures play important roles in the biological functions of genomes and proteins [1]. It is well known that overall about one-third of the whole human genome consists of repeated subsequences [2]; about 10–25% of all known proteins have some form of repetitive structures [3]. In addition, a number of significant problems in molecular sequence analysis can be reduced to repeat finding [4]. Another motivation for finding repeats is to compress the DNA sequences, which is known as one of the most challenging tasks in the data compression field. DNA sequences consist only of symbols from {`ACGT`} and therefore can be represented by two bits per character. Standard compressors such as `gzip` and `bzip` usually use more than two bits per character and therefore cannot reach good compression. Many modern genomic sequence data compression techniques highly rely on the repeat finding in the sequences [5, 6].

The notion of maximal repeat and super maximal repeat [1, 7, 8, 9] captures all the repeats of the whole string in a space-efficient manner, but it does not track the locality of each repeat and thus can not support the finding of repeats that cover a particular string position. In this paper, we propose and study the problem of finding longest repetitive substrings covering any particular string positions. Because any substring of a repeat is also a repeat, the solution to longest repeat queries effectively provides a "stabbing" tool for practitioners for finding most of the repeats that cover particular string positions.

In this paper, we propose an $O(n)$ time and space algorithm that can find the *leftmost* longest repeat of every string position. We view our solution to be optimal in both time and space, because one has to spend $\Omega(n)$ time and space to read and store the input string.

☆Author names are in alphabetical order.

*Corresponding author. Mailing address: 319F CEB, Eastern Washington University, Cheney, WA 99004, USA. Phone: +1 (509) 359-2817. Fax: +1 (509) 359-2215.

Email addresses: `atalay@mit.edu` (Atalay Mert İleri), `okulekci@medipol.edu.tr` (M. Oğuzhan Külekci), `bojianxu@ewu.edu` (Bojian Xu)

2. Preliminary

We consider a **string** $S[1 \ldots n]$, where each character $S[i]$ is drawn from an alphabet $\Sigma = \{1, 2, \ldots, \sigma\}$. A **substring** $S[i \ldots j]$ of S represents $S[i]S[i+1] \ldots S[j]$ if $1 \le i \le j \le n$, and is an empty string if $i > j$. String $S[i' \ldots j']$ is a **proper substring** of another string $S[i \ldots j]$ if $i \le i' \le j' \le j$ and $j' - i' < j - i$. The **length** of a non-empty substring $S[i \ldots j]$, denoted as $|S[i \ldots j]|$, is $j - i + 1$. We define the length of an empty string as zero. A **prefix** of S is a substring $S[1 \ldots i]$ for some i, $1 \le i \le n$. A **proper prefix** $S[1 \ldots i]$ is a prefix of S where $i < n$. A **suffix** of S is a substring $S[i \ldots n]$ for some i, $1 \le i \le n$. A **proper suffix** $S[i \ldots n]$ is a suffix of S where $i > 1$. We say the character $S[i]$ occupies the string **position** i. We say the substring $S[i \ldots j]$ **covers** the kth position of S, if $i \le k \le j$. For two strings A and B, we write $\mathbf{A} = \mathbf{B}$ (and say A is **equal** to B), if $|A| = |B|$ and $A[i] = B[i]$ for $i = 1, 2, \ldots, |A|$. We say A is lexicographically smaller than B, denoted as $\mathbf{A} < \mathbf{B}$, if (1) A is a proper prefix of B, or (2) $A[1] < B[1]$, or (3) there exists an integer $k > 1$ such that $A[i] = B[i]$ for all $1 \le i \le k - 1$ but $A[k] < B[k]$. A substring $S[i \ldots j]$ of S is **unique**, if there does not exist another substring $S[i' \ldots j']$ of S, such that $S[i \ldots j] = S[i' \ldots j']$ but $i \ne i'$. A substring is a **repeat** if it is not unique. A character $S[i]$ is a **singleton**, if it appears only once in S.

Definition 2.1. *For a particular string position $k \in \{1, 2, \ldots, n\}$, the* **longest repeat (LR) covering position k***, denoted as $LR_\mathbf{k}$, is a repeat substring $S[i \ldots j]$, such that: (1) $i \le k \le j$, and (2) there does not exist another repeat substring $S[i' \ldots j']$, such that $i' \le k \le j'$ and $j' - i' > j - i$.*

Definition 2.2. *For a particular string position $k \in \{1, 2, \ldots, n\}$, the* **left-bounded longest repeat (LLR) starting at position** *k, denoted as $\mathbf{LLR_k}$, is a repeat $S[k \ldots j]$, such that either $j = n$ or $S[k \ldots j+1]$ is unique.*

Obviously, for any string position k, if $S[k]$ is not a singleton, both LR_k and LLR_k must exist, because at least $S[k]$ itself is a repeat. Further, there might be multiple choices for LR_k. For example, if $S =$ `abcabcddbca`, then LR_2 can be either $S[1 \ldots 3] =$ `abc` or $S[2 \ldots 4] =$ `bca`. However, if LLR_k does exist, it must have only one choice, because k is a fixed string position and the length of LLR_k must be as long as possible.

The **suffix array** $SA[1 \ldots n]$ of the string S is a permutation of $\{1, 2, \ldots, n\}$, such that for any i and j, $1 \le i < j \le n$, we have $S[SA[i] \ldots n] < S[SA[j] \ldots n]$. That is, $SA[i]$ is the starting position of the ith suffix in the sorted order of all the suffixes of S. The **rank array** $Rank[1 \ldots n]$ is the inverse of the suffix array. That is, $Rank[i] = j$ iff $SA[j] = i$. The **longest common prefix (lcp) array** $LCP[1 \ldots n+1]$ is an array of $n+1$ integers, such that for $i = 2, 3, \ldots, n$, $LCP[i]$ is the length of the lcp of the two suffixes $S[SA[i-1] \ldots n]$ and $S[SA[i] \ldots n]$. We set $LCP[1] = LCP[n+1] = 0$. In the literature, the lcp array is often defined as an array of n integers. We include an extra zero at $LCP[n+1]$ is only to simplify the description of our upcoming algorithms. Table .1 in the appendix shows the suffix array and the lcp array of the example string `mississippi`.

The next Lemma 2.1 shows that, by using the rank array and the lcp array of the string S, it is easy to calculate any LLR_i if it exists or to detect the fact that it does not exist.

Lemma 2.1. *For $i = 1, 2, \ldots, n$:*

$$LLR_i = \begin{cases} S[i \ldots i + L_i - 1] & , \quad \text{if} \quad L_i > 0 \\ \text{does not exist} & , \quad \text{if} \quad L_i = 0 \end{cases}$$

where $L_i = \max\{LCP[Rank[i]], LCP[Rank[i]+1]\}$.

Proof. Note that L_i is the length of the lcp between the suffix $S[i \ldots n]$ and any other suffix of S. If $L_i > 0$, it means substring $S[i \ldots L_i - 1]$ is the lcp among $S[i \ldots n]$ and any other suffix of S. So $S[i \ldots L_i - 1]$ is LLR_i. Otherwise ($L_i = 0$), the letter $S[i]$ is a singleton, so LLR_i does not exist. ☐

Algorithm 1: Find LR_k. Return the leftmost one if k has multiple LRs.

Input: The position index k, and the rank array and the lcp array of the string S

Output: LR_k or find no such LR. The leftmost one will be returned if k has multiple LRs.

```
1 start ← −1; length ← 0 ;                          // start position and length of LR_k
2 for i = k down to 1 do
3     L ← max{LCP[Rank[i]], LCP[Rank[i] + 1]};      // Length of LLR_i
4     if L = 0 or i + L − 1 < k then                // LLR_i does not exist or does not cover k.
5         break;                                    // Early stop
6     else if L ≥ length then                       // Tie is resolved by picking the leftmost one.
7         start ← i; length ← L;
8 Print LR_k ← ⟨start, length⟩;
```

3. Longest repeat finding for one position

In this section, we want to find LR_k for a given string position k, using $O(n)$ time and space. We present the solution to this setting here in case the practitioners have only a smaller number of string positions, for which they want to find the longest repeats, and thus this light-weighted solution will suffice. We will start with finding the leftmost LR_k if the string position k is covered by multiple LRs. In the end of the section, we will show a trivial extension to find all LRs covering position k with the same time and space complexities, if k has multiple LRs.

Lemma 3.1. *Every LR is an LLR.*

Proof. Assume that $LR_k = S[i \dots j]$ is not an LLR. Note that $S[i \dots j]$ is a repeat starting from position i. If $S[i \dots j]$ is not an LLR, it means $S[i \dots j]$ can be extend to some position $j' > j$, so that $S[i \dots j']$ is still a repeat and also covers position k. That says, $|S[i \dots j']| > |S[i \dots j]|$. However, the contradiction is that $S[i \dots j]$ is already the longest repeat covering position k. □

Lemma 3.2. *For any three string positions i, j, and k, $1 \leq i < j \leq k \leq n$: if LLR_j does not exist or does not cover position k, LLR_i does not exist or does not cover position k either.*

Proof. (1) If LLR_j does not exist, then $S[j]$ is a singleton. If LLR_i does exist and covers position k, then LLR_i also covers position j, which yields a contradiction that the substring LLR_i includes the singleton $S[j]$ but is a repeat. (2) If $LLR_j = S[j \dots t]$ does exist but does not cover position k, then $S[j \dots t+1]$ is unique and $t + 1 \leq k$. If LLR_i exists and covers position k, say $LLR_i = S[i \dots r]$, $r \geq k$, it means $S[j \dots t+1]$ is a substring of a repeat $LLR_i = S[i \dots r]$, because $i < j < t+1 \leq r$, so $S[j \dots t+1]$ is also a repeat. This contradicts to the fact that $S[j \dots t+1]$ is unique. So LLR_i does not exist or does not cover position k. □

The idea behind the algorithm for finding the LR covering a given position is straightforward. Algorithm 1 shows the pseudocode, where the found LR is returned as a tuple $\langle start, length \rangle$, representing the starting position and the length of the LR, respectively. If the LR that is being searched for does not exist, $\langle -1, 0 \rangle$ is returned by Algorithm 1. We know that any longest repeat covering position k must be an LLR (Lemma 3.1), starting between indexes 1 to k inclusive. What we need to do is to simply compute every individual of $LLR_1 \dots LLR_k$ using Lemma 2.1 and check whether it covers position k or not. We will just choose the longest LLR that covers position k and resolve the tie by picking the leftmost one if k is covered by multiple LRs (Line 6). Due to Lemma 3.2, a practical speedup is possible via an early stop (Line 5) by computing and checking from LLR_k down to LLR_1 (Line 2).

Lemma 3.3. *Given the rank array and the lcp array of the string S, for any position k in the string S, Algorithm 1 can find LR_k or the fact that it does not exist, using $O(k)$ time and $O(n)$ space. If there are multiple candidates for LR_k, the leftmost one is returned.*

Proof. The algorithm clearly has no more than k steps and each step takes $O(1)$ time, so it costs a total of $O(k)$ time. The space cost is primarily from the rank array and the lcp array, which altogether is $O(n)$, assuming each integer in these arrays costs a constant number of bytes. $\square$

Theorem 3.1. *For any position k in the string S, we can find LR_k or the fact that it does not exist, using $O(n)$ time and space. If there are multiple candidates for LR_k, the leftmost one is returned.*

Proof. The suffix array of S can be constructed by existing algorithms using $O(n)$ time and space (For ex., [10]). After the suffix array is constructed, the rank array can be trivially created using $O(n)$ time and space. We can then use the suffix array and the rank array to construct the lcp array using another $O(n)$ time and space [11]. Given the rank array and the lcp array, the time cost of Algorithm 1 is $O(k)$ (Lemma 3.3). So altogether, we can find LR_k or the fact that it does not exists using $O(n)$ time and space. If multiple LRs cover position k, the leftmost LR will be returned as is guaranteed by Line 6 of Algorithm 1. $\square$

Extension: Find all LRs covering a given position. It is trivial to extend Algorithm 1 to find all the LRs covering any given position k as follows. We can first use Algorithm 1 to find the leftmost LR_k. If LR_k does exist, then we will start over again to recheck LLR_k down to LLR_1 and return those whose length is equal to the length of LR_k. Due to Lemma 3.2, the same early stop as we have in Algorithm 1 can be used for a practical speedup. The pseudocode of this procedure is provided in Algorithm 4 in the appendix, which clearly costs an extra $O(k)$ time. Combining Theorem 3.1, we have:

Theorem 3.2. *We can find all the LRs covering any given position k using $O(n)$ time and space.*

4. Longest repeat finding for every position

In this section, we want to find LR_k of every position $k = 1, 2, \dots, n$. If any position k is covered by multiple LRs, the leftmost one will be returned. A natural solution is to iteratively use Algorithm 1 as a subroutine to find every LR_k, for $k = 1, 2, \dots, n$. However, the total time cost of this solution will be $O(n) + \sum_{k=1}^{n} O(k) = O(n^2)$, where $O(n)$ captures the time cost for the construction of the rank array and the lcp array and $\sum_{k=1}^{n} O(k)$ is the total time cost for the n instances of Algorithm 1. We want to have a solution that costs a total of $O(n)$ time and space, which follows that the amortized cost for finding each LR is $O(1)$.

4.1. A conceptual algorithm

We will first calculate $LLR_1, LLR_2, \dots, LLR_n$ using Lemma 2.1, and save the results in an array $LLRS[1 \dots n]$. Each LLR is represented by a tuple $\langle start, length \rangle$, the starting position and the length of the LLR. We assign zero as the length of any non-existing LLR, which does not cover any string position. We then sort the $LLRS$ array in the descending order of the lengths of the LLRs, using a stable and linear-time sorting procedure such as the counting sort.

Definition 4.1. *After the $LLRS$ array is stably sorted, let P_1 denote the string positions that are covered by $LLRS[1]$, and P_i, $2 \le i \le n$, denote the string positions that are covered by $LLRS[i]$ but are not covered by any of $LLRS[1 \dots i-1]$. Let $|P_i|$ denote the number of string positions belonging to P_i.*

Note that any P_i, $i \ge 1$, can possibly be empty. Our conceptual algorithm will then assign $LLRS[i]$ as the LR of those string positions belonging to P_i, if P_i is not empty, for $i = 1, 2, \dots, n$. We store the LRs that we have calculated in an array $LRS[1 \dots n]$ of $\langle start, length \rangle$ tuples, where $LRS[i] = LR_i$ and $LRS[i].start$ and $LRS[i].length$ represent the starting position and length of LR_i. If LR_i does not exist, the tuple $\langle -1, 0 \rangle$ will be assigned to $LRS[i]$, which can be done during the initialization of the LRS array. Early stop can be made when (1) we meet an $LLRS$ array element whose length is zero, which indicates that all the remaining $LLRS$ array elements also have lengths of zero; or (2) every string position has had their LR calculated. Algorithm 2 shows the pseudocode of this conceptual algorithm.

Algorithm 2: The conceptual algorithm for finding the leftmost LR for every non-singleton string position of S.

Input: The rank array and the lcp array of the string S
Output: The leftmost LR covering every non-singleton string position of S.

/* Calculate the $LLRS$ array using Lemma 2.1. Initialize the LRS array.. */
1 **for** $i = 1, 2, \dots, n$ **do**
2     $LLRS[i] \leftarrow (i, \max\{LCP[Rank[i]], LCP[Rank[i] + 1]\})$; // LLR_i, in the format of $\langle start, length \rangle$
3     $LRS[i] \leftarrow \langle -1, 0 \rangle$; // LR_i, in the format of $\langle start, length \rangle$

4 Stably sort $LLRS[1 \dots n]$ in the descending order of its second dimension ; // e.g.: counting sort.

/* Find the leftmost LR for every position */
5 $count \leftarrow 0$; // The number of non-singleton string positions that have their LRs calculated.
6 **for** $i = 1, 2, \dots, n$ **do**
7   **if** $count = n$ or $LLRS[i].length = 0$ **then** break ; // Early stop
8   **if** $|P_i| = 0$ **then** continue;
9   **foreach** $k \in P_i$ **do** $LRS[k] \leftarrow LLRS[i]$; // Calculate the LRs of the positions belonging to P_i.
10     $count \leftarrow count + |P_i|$;

11 **return** $LRS[1 \dots n]$

Lemma 4.1. *Algorithm 2 finds the LR for every position that does not contain a singleton. It finds the leftmost LR if any position is covered by multiple LRs.*

Proof. The proof of the lemma is obvious. Recall that every LR must be an LLR (Lemma 3.1) and we process all LLRs in descending order of their lengths. For $i = 1, 2, \dots, n$, if P_i is not empty, then for each position in P_i, the substring $LLRS[i]$ is the longest LLR that covers that position, i.e., $LLRS[i]$ is the LR of that position. In the case where any position in P_i has multiple LRs, $LLRS[i]$ must be the leftmost LR because of the stable sorting of the $LLRS$ array. □

4.2. High-level strategy for a fast implementation

The challenge is to implement the conceptual algorithm (Algorithm 2) efficiently. Our goal is to use $O(n)$ time and space only, which is optimal since we have to spend $O(n)$ time and space to report all the LRs of all the n distinct string positions. We start with some property of each P_i (Definition 4.1). Recall that, in Algorithm 2, we process all the LLRs in the descending order of their lengths, and also all LLRs start from distinct string positions. Therefore, after the $LLRS$ array is sorted (Line 4, Algorithm 2), none of $LLRS[1 \dots i-1]$ can be a substring of $LLRS[i]$, for any $i \geq 2$. This yields the following fact.

Fact 4.1. *Every non-empty P_i, $i \geq 1$, is a continuous chunk of string positions, i.e., every non-empty P_i is an integer range $[s_i, e_i]$, where s_i and e_i are the starting and ending string positions of P_i.*

In the case where P_i is empty, we set $s_i = e_i = -1$. In order to achieve an overall $O(n)$-time implementation of Algorithm 2, we need a mechanism that can quickly find s_i using $O(1)$ time when processing each $LLRS[i]$. Then, if $s_i \neq -1$, due to Fact 4.1, we can just linearly walk from string position s_i through the position e_i, which is either the right boundary of $LLRS[i]$ or a string position whose next neighboring position has had its LR calculated, whichever one is reached first. We will then set the LR of each visited position during the walk to be $LLRS[i]$, achieving an overall $O(n)$ time implementation of Algorithm 2.

When we process a non-empty $LLRS[i]$ and calculate its s_i, there are two cases. Case 1: The string position $LLRS[i].start$ has not had its LR calculated, then obviously $s_i = LLRS[i].start$. Case 2: The string position $LLRS[i].start$ has already had its LR calculated, then it is either $s_i > LLRS[i].start$ (if P_i is not empty) or $s_i = -1$ (if P_i is empty). In this case, it will not be efficient to find s_i by simply walking from $LLRS[i].start$ toward e_i until we reach e_i or a string position whose LR has not been calculated. It is not immediately clear how to calculate s_i using $O(1)$ time. This leads to the design of our following mechanism that enables us to calculate every s_i in Case 2 using $O(1)$ time.

4.3. The two-table system: the ptr and next arrays

Our mechanism is built upon two integer arrays, $ptr[1 \dots n]$ and $next[1 \dots n]$. We update the two arrays online when we process the sorted $LLRS$ array elements in the calculation of the LR of every string position. *Ideally*, we want to maintain these two arrays, such that for any string position k that has had its LR calculated, $next\big[ptr[k]\big]$ is either the next after-k string position whose LR is not calculated yet or $n+1$ if no such after-k string position exists. Then, when we process a particular non-empty $LLRS[i]$, if the string position $LLRS[i].start$ has had its LR calculated, we can either directly get s_i or find the fact that all string positions covered by $LLRS[i]$ have had their LRs calculated, by comparing $next\Big[ptr\big[LLRS[i].start\big]\Big]$ and $LLRS[i].start + LLRS[i].length - 1$ (the right boundary of $LLRS[i]$). However, it is not clear how to achieve such an ideal maintenance of the *ptr* and *next* arrays in a time-efficient manner. This motivates us to maintain these two arrays *approximately*, which is to maintain the following invariance. We will show later that such approximate maintenance of the *ptr* and *next* arrays can still help calculate every s_i using $O(1)$ time.

4.3.1. Invariance.

We initialize every element of both *ptr* and *next* arrays to be -1. Recall that after the $LLRS$ array is sorted in descending order of the lengths of the LLRs, we process every $LLRS[i]$, for $i = 1, 2, \dots, n$. After we have finished the processing of $LLRS[1 \dots i-1]$, for any $i \geq 2$, we want to maintain the following invariance for the *ptr* and *next* arrays when processing $LLRS[i]$.

1. If $LLRS[i].start$ has already had its LR calculated but $|P_i| > 0$, then:

$$next\Big[ptr\big[LLRS[i].start\big]\Big] = s_i$$

2. If $|P_i| = 0$ but $LLRS[i].length > 0$ (i.e.: $LLRS[i]$ is not empty), then $next\Big[ptr\big[LLRS[i].start\big]\Big]$ is larger than the index of the right boundary of $LLRS[i]$. That is,

$$next\Big[ptr\big[LLRS[i].start\big]\Big] > LLRS[i].start + LLRS[j].length - 1$$

4.3.2. Using the invariance.

Recall that when we process a particular non-empty $LLRS[i]$, we want to calculate s_i quickly. The hard case is when the string position $LLRS[i].start$ has already had its LR calculated. Provided with the above invariance of the *ptr* and *next* arrays, when we process a non-empty $LLRS[i]$, we will first check the value of $ptr\big[LLRS[i].start\big]$. If it is not equal to -1, the hard case occurs. Then, if $next\Big[ptr\big[LLRS[i].start\big]\Big] \leq LLRS[i].start + LLRS[i].length - 1$ (the right boundary of $LLRS[i]$), we can assert $s_i = next\Big[ptr\big[LLRS[i].start\big]\Big]$; otherwise, we can assert P_i is empty and thus will simply skip $LLRS[i]$.

4.4. Maintaining the two-table system.

In the following, we will first describe how we update the *ptr* and *next* arrays when processing every $LLRS[i]$. In the end, we will explain why the invariance is maintained using an overall $O(n)$ time. Remind that the whole algorithm will early stop if $LLRS[i]$ is empty, so we will only need to describe the algorithmic for processing a non-empty $LLRS[i]$. We first initialize every element in both *ptr* and *next* arrays to be -1. We will use the word *bucket* to denote a maximal and continuous area in the *ptr* array where all entries share the same positive value. So initially, there is no bucket presented in the *ptr* array. Because all the $LLRS$ array elements have been sorted in the descending order of their lengths, the maintenance of the two-table system will only have the following five cases to consider (Figure 1). We use $left$ and $right$ to denote the indexes of the left and right boundary of the $LLRS[i]$. That is,

```
left  ← LLRS[i].start; // the left boundary of LLRS[i]
right ← LLRS[i].start + LLRS[i].length - 1; // the right boundary of LLRS[i]
```

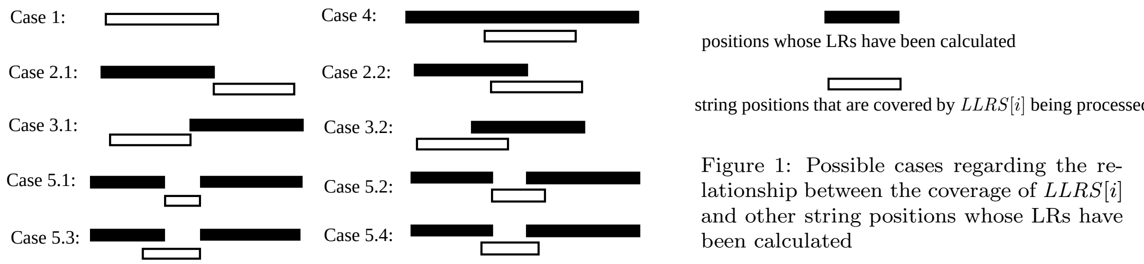

Figure 1: Possible cases regarding the relationship between the coverage of $LLRS[i]$ and other string positions whose LRs have been calculated

Case 1: The coverage of $LLRS[i]$ does not connect to or overlap with any string positions whose LRs have been calculated. We will create a bucket in the *ptr* array covering the string positions that are covered by $LLRS[i]$ and set up the corresponding *next* array entry to be the string position that is right after the coverage of $LLRS[i]$. The following code shows the case condition and the update made to the *ptr* and *next* arrays.

```
if ptr[left] = -1 and ptr[right] = -1 and (left = 1 or ptr[left-1] = -1)
                  and (right = n or ptr[right+1] = -1)  //case 1
   for j = left...right: ptr[j] ← i;
   next[i] ← right + 1;
```

Case 2: The coverage of $LLRS[i]$ connects to or overlaps with the right side of a string position area whose LRs have been calculated. We will extend that area's corresponding *ptr* array bucket to the coverage of $LLRS[i]$ and update the corresponding *next* array entry to be the string position that is right after the new bucket.

```
else if (ptr[left] = -1 and left ≠ 1 and ptr[left-1] ≠ -1) and ptr[right] = -1
                        and (right = n or ptr[right+1] = -1) //case 2.1
   for j = left...right: ptr[j] ← ptr[left-1];
   next[ptr[left-1]] ← right + 1;

else if (ptr[left] ≠ -1) and ptr[right] = -1
                         and (right = n or ptr[right+1] = -1)  //case 2.2
   for j = next[ptr[left]]...right: ptr[j] ← ptr[left];
   next[ptr[left]] ← right + 1;
```

Case 3: The coverage of $LLRS[i]$ connects to or overlaps with the left side of an existing string position area whose LRs have been calculated. We will left-extend that area's corresponding *ptr* array bucket to the coverage of $LLRS[i]$. We need not to update the corresponding *next* array entry, since the string position that is right after the new *ptr* bucket does not change.

```
else if (ptr[right] = -1 and right ≠ n and ptr[right+1] ≠ -1) and ptr[left] = -1
                         and (left = 1 or ptr[left-1] = -1) //case 3.1
   for j = left...right: ptr[j] ← ptr[right+1];

else if (ptr[right] ≠ -1) and ptr[left] = -1
                          and (left = 1 or ptr[left-1] = -1) //case 3.2
   j ← left;
   while ptr[j] = -1: ptr[j] ← ptr[right]; j++;
```

Case 4: Every string position covered by $LLRS[i]$ has its LR calculated already. In this case, we simply do nothing.

```
else if ptr[left] ≠ -1 and next[ptr[left]] > right: do nothing; //case 4
```

Case 5: The coverage of $LLRS[i]$ bridges two string position areas whose LRs have been calculated. We will extend the left area's corresponding *ptr* array bucket up to the left boundary of the right area and update the *next* array entry of the left area to be the one of the right area.

```
else
   if ptr[left] = -1: j ← left; ptr_entry ← ptr[left-1]; //case 5.1, 5.2
   else:              j ← next[ptr[left]]; ptr_entry ← ptr[left]; //case 5.3, 5.4
   while ptr[j] = -1: ptr[j] ← ptr_entry; j++;
   next[ptr[ptr_entry]] ← next[ptr[j]];
```

Lemma 4.2 (Correctness)**.** *The two-table system's invariance is maintained.*

Proof Sketch: Let us call a *ptr* array bucket $ptr[i \ldots j]$ as a *tail bucket* if $j = n$ or $ptr[j+1] = -1$. (1) We first prove that the invariance is maintained on tail buckets. Observe that any tail *ptr* bucket is created in Case 1 (Figure 1) and can be extended in Case 2 as well as in Case 3 if the black bucket in Case 3 was also a tail bucket. The update to the tail bucket as well as the corresponding *next* array entry guarantees that, for any i belonging to the coverage of a tail bucket, $next\big[ptr[i]\big]$ is *ideally* equal to the index of the string position that is right after the bucket (or $n+1$ if no such string position exists). So obviously the invariance is maintained. (2) We now prove the invariance is also maintained on non-tail buckets. Observe that any non-tail bucket is created in Case 5 from the merge of the left black bucket and the new $LLRS[i]$'s coverage. After such non-tail bucket is created, for any position i belonging to a non-tail bucket, $next\big[ptr[i]\big]$ is at least as large as the index of the string position that is following the right black bucket in Case 5. That means $next\big[ptr[i]\big] - i$ is larger than the size of any unprocessed *llrs* array element. This guarantee is maintained, because every $next\big[ptr[i]\big]$ only monotonically increases. So, the invariance is also maintained for non-tail buckets. (3) Because the invariance is well maintained for all *ptr* buckets, it is safe to have the condition checking as we have written for Case 4. ■

Lemma 4.3 (Time complexity)**.** *The two-table system is maintained using a total of $O(n)$ time over the course of the processing of the LLRS array elements.*

Proof Sketch: Observe that the updates to the *ptr* array are made only to those entries whose values were -1 and the new values from the updates are all positive. So there are no more than n updates to the *ptr* array. It is obvious that the number of updates made to the *next* array is no more than the number of updates made to the *ptr* array. Other than *ptr* and *next* array updates, the rest of the maintenance work for the two-table system when processing each *LLRS* array element takes $O(1)$ time. So the total time cost in maintaining the two-table system over the course of the processing of the whole *LLRS* array is $O(n)$. ■

4.5. The final $O(n)$ time and space algorithm.

By combining the conceptual Algorithm 2, the high-level strategy for the fast implementation, and the two-table system's maintenance mechanism, we are ready to produce the final $O(n)$ time and space algorithm that can find the leftmost LR of every string position. Algorithm 3 shows the pseudocode. It starts with the calculation of the *LLRS* array and the initialization of the *LRS*, *ptr*, and *next* arrays (Line 1–4). It then sorts the *LLRS* array in the descending order of the lengths of the array elements using a linear and stable sorting procedure (Line 5). It then uses the `for` loop (Line 7) to process every *LLRS* array element with possible early stop (Line 8). Using the two-table system, the value of s_i is calculated by Line 10 if P_i is not empty; otherwise, the fact that P_i is empty will also be detected by Line 11. After s_i is calculated, finding the LR of each position in P_i becomes obvious (Line 12–14). After the LR finding work is done, we will update the two-table system (Line 15) using the code presented in Section 4.4.

Lemma 4.4. *Given the lcp array and the rank array, Algorithm 3 calculates the leftmost LR of every non-singleton position of a string S of size n using a total $O(n)$ time and space.*

Proof Sketch: (1)*Correctness.* The correctness of Algorithm 3 immediately follows from of Lemma 4.1 and Lemma 4.2. (2) All data structures that are being involved are the *LCP*, *Rank*, *LLRS*, *LRS*, *ptr*, and *next* arrays. Altogether they use $O(n)$ space. (3) The time cost for the initialization (Line 1–4) takes $O(n)$ time. the stable sorting (Line 5) uses $O(n)$ time. The rest of the work (Line 7–15) also takes $O(n)$ time, because we update every *LRS* array element no more than once and the two-table system maintenance also takes $O(n)$ time (Lemma 4.3). So the total time cost is $O(n)$. ■

Theorem 4.1. *Given a string S of size n, we can calculate the leftmost LR of every of string position using $O(n)$ time and space.*

Algorithm 3: The $O(n)$ time and space algorithm for finding the leftmost LR for every non-singleton string position of S.

Input: The rank array and the lcp array of the string S

Output: The leftmost LR covering every non-singleton string position of S.

```
/* Calculate the LLRS array using Lemma 2.1.
   Initialize the LRS array and the auxiliary ptr and next arrays.                         */
for i = 1, 2, ..., n do
    LLRS[i] ← ⟨i, max{LCP[Rank[i]], LCP[Rank[i] + 1]}⟩ ;    // LLR_i, in the format of ⟨start, length⟩
    LRS[i] ← ⟨−1, 0⟩ ;                                     // LR_i, in the format of ⟨start, length⟩
    ptr[i] ← −1;   next[i] ← −1 ;
Stably sort LLRS[1 . . . n] in the descending order of its second dimension ;   // e.g.: counting sort.

/* Find the leftmost LR for every position                                                 */
count ← 0 ;   // The number of non-singleton string positions that have their LRs calculated.
for i = 1, 2, ..., n do
    if count = n or LLRS[i].length = 0 then break ;                                 // Early stop
    left ← LLRS[i].start; right ← LLRS[i].start + LLRS[i].length − 1; // The boundaries of LLRS[i].
    /* first = s_i of P_i = [s_i, e_i] if P_i is not empty.                                 */
    if ptr[left] = −1 then first ← left; else first ← next[ptr[left]];
    if first > right then continue ;                           // Detect the fact that P_i is empty.
    /* Calculate the the leftmost LR of every position in P_i = [s_i, e_i].                 */
    j ← first;
    while j ≤ right and ptr[j] = −1 do
        LRS[j] ← ⟨LLRS[i].start, LLRS[i].length⟩; count ← count + 1; j ← j + 1;
    Update the two-table system here using the code presented in Section 4.4.
return LRS[1 . . . n]
```

Proof Sketch: We can construct the suffix array of the string S in a total of $O(n)$ time and space using existing algorithms (For ex., [10]). The rank array is just the inverse suffix array and can be directly obtained from SA using $O(n)$ time and space. Then we can obtain the lcp array from the suffix array and rank array using another $O(n)$ time and space [11]. So the total time and space costs for preparing the rank and lcp arrays are $O(n)$. The proof of the theorem can then immediately follow from Lemma 4.4. ■

5. Conclusion

In this paper, we proposed the problem of finding longest repeats covering particular string positions, motivated by its applications in subfields such as computational biology. We proposed optimal algorithms for finding the (leftmost) longest repeat of every string position using a total of $O(n)$ time and space based on a novel two-table system that we designed. We have implemented our algorithms. Future work can be an experimental study of the implementation.

References

[1] D. Gusfield, Algorithms on strings, trees and sequences: computer science and computational biology, Cambridge University Press, 1997.

[2] E. H. McConkey, Human Genetics: The Molecular Revolution, Jones and Bartlett, Boston, MA, 1993.

[3] X. Liu, L. Wang, Finding the region of pseudo-periodic tandem repeats in biological sequences, Algorithms for Molecular Biology 1 (1) (2006) 2.

[4] H. M. Martinez, An efficient method for finding repeats in molecular sequences, Nucleic Acids Res. 11 (13) (1983) 4629–4634.

[5] G. Manzini, M. Rastero, A simple and fast dna compressor, Software – Practice and Experience 34 (2004) 1397–1411.

[6] B. Behzadi, F. L. Fessant, Dna compression challenge revisited: A dynamic programming approach, in: Annual Symposium on Combinatorial Pattern Matching, 2005.

[7] M. O. Kulekci, J. S. Vitter, B. Xu, Time- and space-efficient maximal repeat finding using the burrows-wheeler transform and wavelet trees, in: IEEE International Conference on Bioinformatics and Biomedicine, 2010, pp. 622–625.

[8] M. O. Kulekci, J. S. Vitter, B. Xu, Efficient maximal repeat finding using the burrows-wheeler transform and wavelet tree, IEEE Transactions on Computational Biology and Bioinformatics (TCBB) 9 (2) (2012) 421–429.

[9] T. Beller, K. Berger, E. Ohlebusch, Space-efficient computation of maximal and supermaximal repeats in genome sequences, in: Proceedings of the 19th International Conference on String Processing and Information Retrieval (SPIRE), 2012, pp. 99–110.

[10] P. Ko, S. Aluru, Space efficient linear time construction of suffix arrays, Journal of Discrete Algorithms 3 (2-4) (2005) 143–156.

[11] T. Kasai, G. Lee, H. Arimura, S. Arikawa, K. Park, Linear-time longest-common-prefix computation in suffix arrays and its applications, in: Symposium on Combinatorial Pattern Matching, 2001, pp. 181–192.

Appendix

i	$LCP[i]$	$SA[i]$	suffixes
1	0	11	`i`
2	1	8	`ippi`
3	1	5	`issippi`
4	4	2	`ississippi`
5	0	1	`mississippi`
6	0	10	`pi`
7	1	9	`ppi`
8	0	7	`sippi`
9	2	4	`sissippi`
10	1	6	`ssippi`
11	3	3	`ssissippi`
12	0	–	–

Table .1: The suffix array and the lcp array of an example string $S =$ `mississippi`.

Algorithm 4: Find all LRs that cover a given position k

Input: The position index k, and the rank array and the lcp array of the string S
Output: All LRs that cover position k or find no such LR.

`/* Find the length of` LR_k. `*/`
$length \leftarrow 0$;
for $i = k$ *down to* 1 **do**
    $L \leftarrow \max\{LCP[Rank[i]], LCP[Rank[i]+1]\}$; `//` `Length of` LLR_i
    if $L = 0$ *or* $i + L - 1 < k$ **then** `//` LLR_i `does not exist or does not cover` k.
        break; `// Early stop`
    else if $L \geq length$ **then**
        $length \leftarrow L$;

`/* Print all LRs that cover position` k. `*/`
if $length > 0$ **then** `//` LR_k `does exist.`
    for $i = k$ *down to* 1 **do**
        $L \leftarrow \max\{LCP[Rank[i]], LCP[Rank[i]+1]\}$; `//` `Length of` LLR_i
        if $L = 0$ *or* $i + L - 1 < k$ **then** `//` LLR_i `does not exist or does not cover` k.
            break; `// Early stop`
        else if $L = length$ **then**
            Print $LR_k \leftarrow \langle i, length \rangle$;
else Print $LR_k \leftarrow \langle -1, 0 \rangle$; `//` LR_k `does not exist.`